UFR Sciences et Techniques de Besançon

Project Master SIS — 2nd year

Long range wireless sensor communication using LoRaWAN relays.

Author :
Arthur Matusalem

Supervisor :
Jean-Michel Friedt

2022 – 2023

Contents

1 Introduction **3**

2 LoRaWAN Communication protocol **3**

2.1 Chirp Spread Spectrum modulation . . . 3
2.2 LoRa and LoRaWAN . . . 4
2.3 LoRaWAN packet format . . . 5
2.4 Frequency band . . . 7
2.5 Devices classes . . . 7
2.6 Typical communication procedure . . . 8

3 RIOT **9**

4 Simple communication from endpoint to the gateway **9**

4.1 Decoding messages . . . 10

5 Reducing the endpoint communication to one frequency **12**

5.1 Introduction . . . 12
5.2 Use a mask for the channels . . . 12
5.3 Delete the channels definitions in header files . . . 12
5.4 Change the way to choose the next channel of communication . . . 13

6 Implementing the relay **14**

6.1 Receive a message from endpoint on the relay . . . 14
6.2 Sending a message from the relay and receive it on the endpoint . . . 16
6.3 Communication from endpoint to gateway passing by the relay . . . 16

7 Conclusion **18**

1 Introduction

"In the context of long term Arctic glacier monitoring, an array of temperature probes has been distributed on Austre Lovenbreen, a glacier in Spitsbergen, 79 deg. N. Since these temperature probes often lie buried under snow in Spring, their battery is changed every year in Autumn despite the chances that they could last another year. Preventive maintenance is impossible without continuous monitoring of the battery level, and data loss occurs when battery levels drop unexpectedly."[1]

This project aims to add a wireless communication to send the level of the battery and to prevent the change of the battery if not needed. In order to do this, we will use LoRaWAN and the LoRa modulation to achieve a long range communication. LoRaWAN will be implemented using the RIOT OS environment that allows us to do the LoRaWAN communication.

In our setup the distance between the emitter and receiver is too large and they are separated by obstacles such as mountains blocking the radiofrequency link, so we need to implement a relay in order to transmit the communication. This is the goal of our project.

The objectives of this project are to:

- become familiar with LoRaWAN, the LoRa modulation and the RIOT OS environment;
- successfully send a LoRaWAN packet between the client (sensor) and the server (basestation) and decode it;
- design a relay with an LoRaE5 card and achieve the communications between all elements.

2 LoRaWAN Communication protocol

2.1 Chirp Spread Spectrum modulation

The principle is instead of having a high datarate communication, we can have a better range of communication by decreasing the bandwidth and hence the datarate. So LoRa modulation is low datarate but long range.

Chirp Spread Spectrum or CSS modulation is used by LoRa to modulate its signals.

The principle is to encode information using chirp pulses. These pulses are a sweep of frequency like we can see on the figure 1 with each chirp duration defined by a Spreading Factor (SF) parameter and a given bandwidth from a set of possible options (125, 250 or 500 kHz).

The chirp frequency increases or decreases depending on the direction of the transmission. If the transmission is "up" (from endpoint to gateway) the frequency increases, if the transmission is "down" (from gateway to endpoint) the frequency decreases.

We will need to pay attention to this information to make our relay because it will need to communicate up and down.

[1]Jean-Michel Friedt

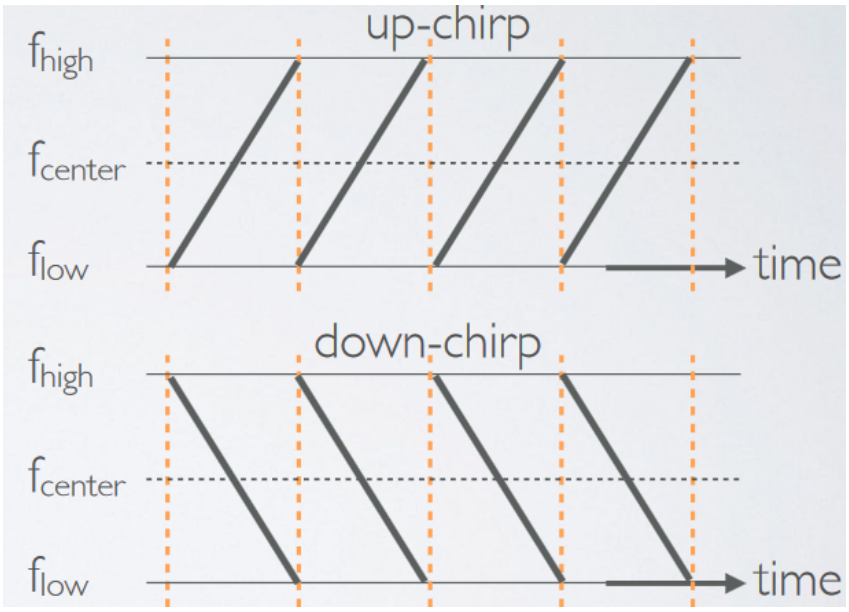


Figure 1: Chirp spread spectrum modulation (CSS) used by LoRa scheme (source : https://Lora.readthedocs.io/en/latest/).

2.2 LoRa and LoRaWAN

LoRaWAN is designed to be a long range and low power protocol based on the LoRa technology.

As we can see on the figure 2, LoRaWAN is an upper layer of LoRa and defines the m Medium Access Control (MAC) layer while LoRa defines the physical layer.

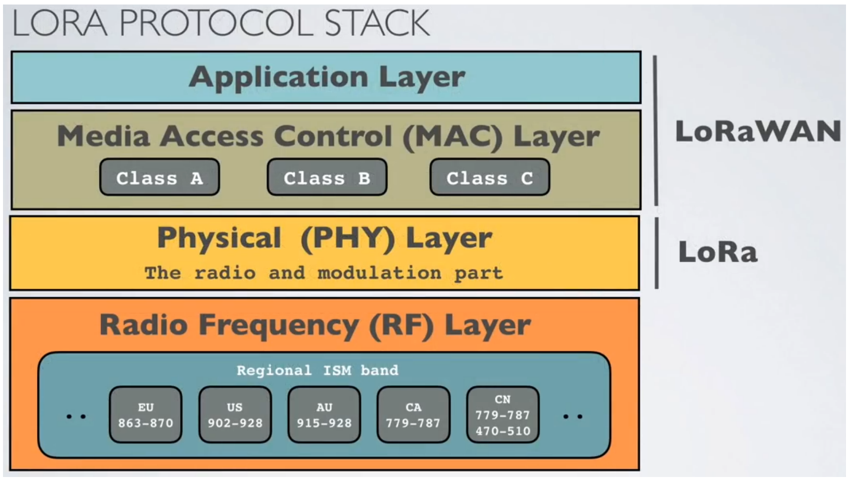


Figure 2: LoRaWAN protocol stack (source : https://LoRa.readthedocs.io/en/latest/).

The principle of LoRaWAN communication is represented on the figure 3.

LoRa communication is always initiated by the endpoint (low power sensor node) towards the gateway (server with little power saving consideration). Once the wireless link from endpoint to gateway using LoRa is achieved, heterogeneous internet connectivity (wireless or wired) is used to route information to a network server. The whole communication from endpoint to network server is encrypted in LoRaWAN: only the network server is informed of the cryptographic keys needed to decipher the transmitted messages.

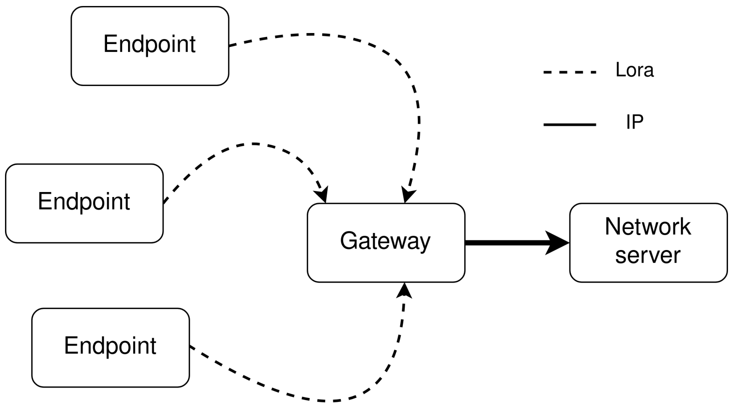


Figure 3: LoRaWAN communication scheme.

Our project is a LoRaWAN relay that listens to endpoints and transmits to the gateway all in LoRa modulation as we can see on the figure 4. A gateway can listen at many frequencies at a time but our relay is based on an endpoint which only listens to one frequency at a time. It means that our relay cannot listen to all endpoints simultaneously.

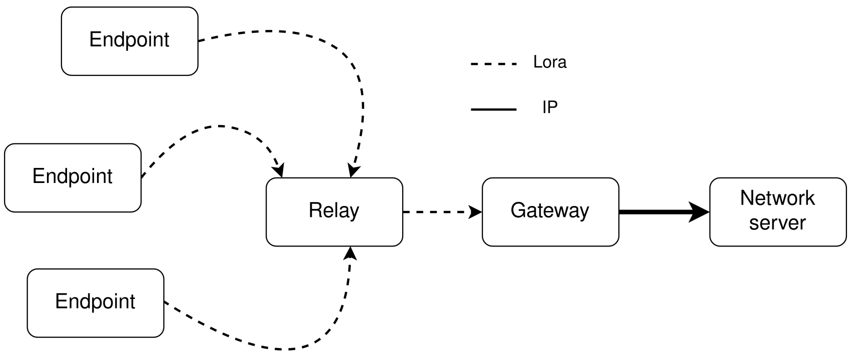


Figure 4: Our project of LoRaWAN relay communication scheme.

A LoRaWAN relay is not a new idea, the developers of LoRaWAN are aware of the need for a relay. We can find some documents that relate standards for a relay[2][3].

But despite the specification of a relay, there is still no implementation of any LoRaWAN relay.

This subject of LoRaWAN continues to interest people as we can see new papers released like the one about LoRaLitE[4]

2.3 LoRaWAN packet format

LoRaWAN packets follow the format of figure 5

[2]LoRaWAN relay specification : https://resources.LoRa-alliance.org/technical-specifications/ts011-1-0-0-relay

[3]Dominique Barthel presentation about LoRaWAN relay specifications at Eclipse IoT Day Grenoble 2023 : https://wiki.eclipse.org/images/7/7a/Eid2023_barthel.pdf, accessed in January 2023

[4]LoRaLitE: LoRa protocol for Energy-Limited environments by Lukasz Sergiusz Michalik, Loic Guegan, Issam Raïs, Otto Anshus and John Markus Bjørndalen, in Proc. IEEE MASCOTS (2022)

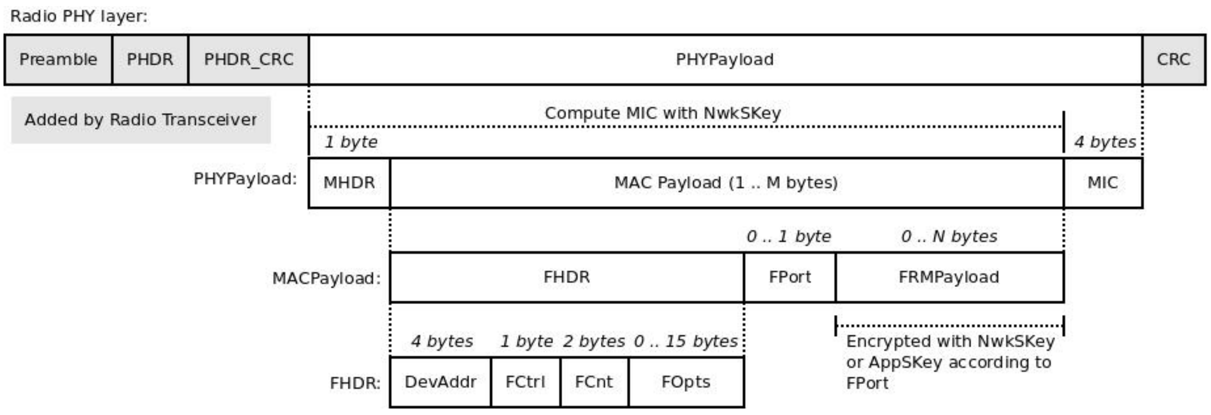


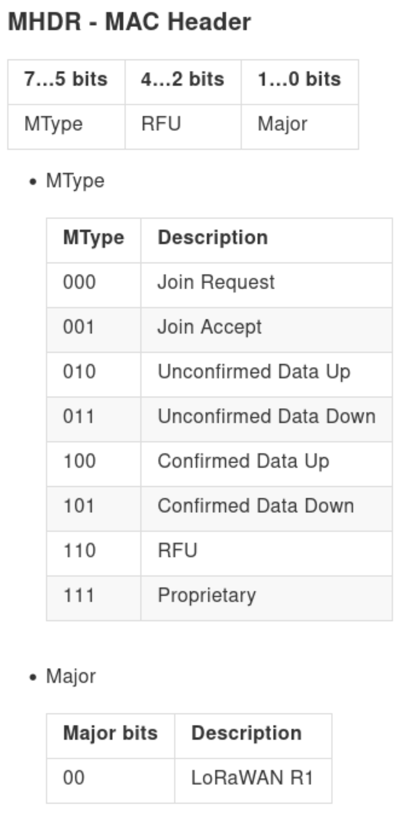
Figure 5: LoRaWAN packet format. (source : https://hackmd.io/@starnight/S1kg6Ymo-).

The information to be transmitted is in the frame payload (FRMpayload on figure 5)

`Fcnts` correspond to the frame counter, it counts the number of packets that has been sent. This number must be synchronised between the endpoint and the gateway

The `MHDR` is the MAC header and contains useful information such as `Mtype`.

`Mtype` indicates the type of messages for example when the connection is established, the message from the endpoint always begin by *0x80*. The 3 most significant bytes are 100 which correspond to *confirmed data up* according to the figure 6 which is consistent with the message sent.



MHDR - MAC Header

7...5 bits	4...2 bits	1...0 bits
MType	RFU	Major

- MType

MType	Description
000	Join Request
001	Join Accept
010	Unconfirmed Data Up
011	Unconfirmed Data Down
100	Confirmed Data Up
101	Confirmed Data Down
110	RFU
111	Proprietary

- Major

Major bits	Description
00	LoRaWAN R1

Figure 6: `MHDR` content and signification table of the three `Mtype` bits. (source : https://hackmd.io/@starnight/S1kg6Ymo-).

We can see on the figure 5 that we have 4 bytes of MIC, this is Message Integrity Code. This is used to verify that the decryption of the message is correct.

We can also see that a message contains the `devAddr`, which is the device address attributed by the network during the join procedure (we also find it in the join accept message.)

We also have a CRC (Cyclic Redundancy Check) to check if the message contains the good number of bits.

Finally, something that is rarely mentioned is the `syncword` which is after the preamble, which will have to be tuned in our application. The syncword is used to differentiate end devices from different applications. For example, if we have an endpoint with a `syncword` of 1 and if the gateway sends a message with a `syncword` of 2, the endpoint will see it but because syncwords are not the same, it will stop receiving the following of the message.

The syncword can be changed in the parameters like the bandwidth or the spreading factor. But LoRaWAN developers suggest to only use two syncwords :

- 0x12 for private networks;
- 0x34 for public networks (the most used);

We use the public syncword for our application but it can maybe be interesting for our relay to differentiate the communication to one endpoint or another. We should make some tests to see if we can use others syncwords.

2.4 Frequency band

LoRa uses frequency hopping when endpoints communicate with a gateway, as mentioned with: "An end device changes channel in a pseudo-random fashion for every transmission. Changing frequencies makes the system more robust to interferences. For example in Europe for uplink transmissions 8 different frequencies are used."[5]

An endpoint communicates with the gateway using many possible frequencies. The frequency used depends on the region the gateway is situated in. For example in the region EU868 (Europe), there are 8 frequencies between 863MHz and 870MHz.

As indicated by the citation above, the endpoint changes the communication frequency randomly when initiating each communication, but remains within the same frequency range for a given message transmission. This will be a problem because if we want to make a relay from a simple LoRaE5 endpoint card, this low-power circuit cannot listen at many channels simultaneously like a gateway. If we cannot determine the next channel frequency, the relay cannot listen to the good channel and will lose some communications.

2.5 Devices classes

There are three classes of LoRaWAN devices: class A, B and C.

The class A is the lowest energy consumption, it will open after the emission of a packet two windows of reception RX1 and RX2.

The class B is a tradeoff between the class A and C. It will open a reception window periodically after an emission of a packet.

Class C will listen continuously after the emission of a packet but it is the one that consumes the most energy.

Compliance of the relay communication with device class regulations might be a challenge since the duty cycle defining the mean power consumption is regulated by the LoRaWAN standard. At the moment we will not aim at compliance with any duty cycle regulation.

[5]https://LoRa.readthedocs.io/en/latest/

2.6 Typical communication procedure

A communication sequence in LoRaWAN between an endpoint and a gateway is done in many steps. These steps are summarized on figure 7

The first thing to begin is the endpoint which must send a join request to the gateway. In this join request, we found in the physical payload seen before on figure 5 the appEUI and the devEUI.

The devEUI is used to identify the device and the appEUI is used to identify the receiver device that can process the join request.

If the join request is accepted by the server, the gateway will send a join accept message to the endpoint that sent the request. This join accept message contains information provided by the server through the gateway for example the Rxdelay used by the endpoint to know how long it must wait for a response before declaring that it failed. It also contains the CFList (Channel Frequency List) which indicates to the endpoint which frequency to use.

Once the join procedure is done, the endpoint can send a packet with some data.

Each time a packet is sent the uplink counter of the endpoint is increased and each time a packet is received, the uplink counter of the gateway is increased. This uplink counter is used to make sure that all and only the messages from the endpoint are received. If these two values are not synchronised, the endpoint will stop to send data. This feature is useful if we do not want someone to interfere with our data.

For our project, we disabled the uplink counter verification for the endpoint to continue sending packets periodically even if the packets are not received on the gateway side.

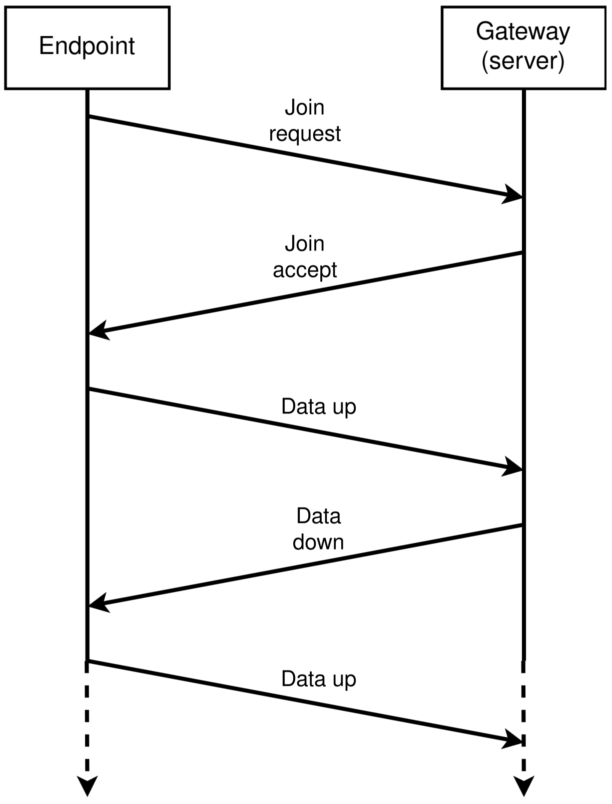


Figure 7: Basic LoRaWAN exchange between endpoint and gateway.

3 RIOT

For this project we will use RIOT OS. RIOT is a free and open source operating system found at `https://github.com/RIOT-OS/RIOT`. RIOT OS supports microcontrollers and especially the STM32 of our LoRaE5 board. It is intended to broaden the audience of developers of embedded systems in the context of the Internet of things (IoT).

RIOT OS allows to implement LoRaWAN for our application. It is a very rich environment with a lot of functions and a good documentation.

As an illustration of the ease of using RIOT OS, configuring a LoRaWAN interface is limited to calling a pair a functions

We have for example two functions of join and send, it will be helpful for our endpoint to join the server and to send messages. But these function have to be used in a specific way.

```
semtech_LoRamac_join(&LoRamac, LoRaMAC_JOIN_OTAA)
semtech_LoRamac_send(&LoRamac, (uint8_t *)message, strlen(message))
```

We have for example tested the sx126x driver to send and receive a message using the driver test program of RIOT found in the `tests/driver_sx126x` directory of the RIOT OS files. Most peripheral use is illustrated by an example in the `tests` directory

We have access to several functions like for example to set plenty of parameters (bandwidth, spreading factor... we can see them on the listing below) and it will allows us to send and receive messages for our relay.

It also has functions to set the different cryptographic keys of the device for example as we can see in the next listing :

```
semtech_LoRamac_set_deveui(LoRamac, deveui);
semtech_LoRamac_set_appeui(LoRamac, appeui);
semtech_LoRamac_set_appkey(LoRamac, appkey);
```

RIOT has a high abstraction level which is useful to have easily access to a LoRaWAN application, without RIOT it would have been very complicated to do a simple communication in LoRaWAN.

On the other hand, when we must do specific things such as sending a message without the join procedure described earlier, it is very hard to find how to do this because it is not designed for that.

But we always have a low level access. For example to change the spreading factor we can use the following line :

```
netdev->driver->set(netdev,NETOPT_SPREADING_FACTOR,&sf, sizeof(uint8_t));
```

We also have other common functionalities like a watchdog for example :

```
wdt_start()
wdt_setup_reboot(0, 5000)
wdt_kick()
```

In this project we have spent a lot of time to search functionalities that was not intended by RIOT and we had to find and modify the header files to achieve what we wanted.

4 Simple communication from endpoint to the gateway

For this first step, we based our code on the program of Didier Donsez[6].

[6] Didier Donsez, Orbimote, https://github.com/CampusIoT/orbimote

We spent a lot of time to clearly understand how it works and how to send something.

To generate the identification and cryptographic keys, we keep the function initially provided by Didier Donsez.

We discovered that the RIOT function of `send` requires that the device has already joined the network with the RIOT function of `join` (See the part on RIOT for these functions).

But for our relay we need to send a message without joining the server. So it cannot be used for our relay function of sending. However, we can use it for our endpoint program.

The function of RIOT already does what is needed to receive the join accept and the down messages.

With our program, we are able to communicate with the gateway. On the network server we can see that we successfully joined and that messages are received.

4.1 Decoding messages

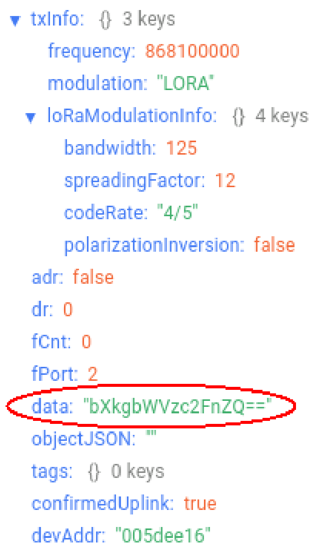
One of the advantages of LoRaWAN compared to just the LoRa modulation is that the data is encrypted, meaning that if someone listens to the same frequency, the message cannot be decrypted without the secret key.

This is an advantage for a functional application but in our case it implies an additional step to recover the transmitted message.

Now that the packets are received, the most important thing is to be able to understand it. From the endpoint we send the message: "my message"

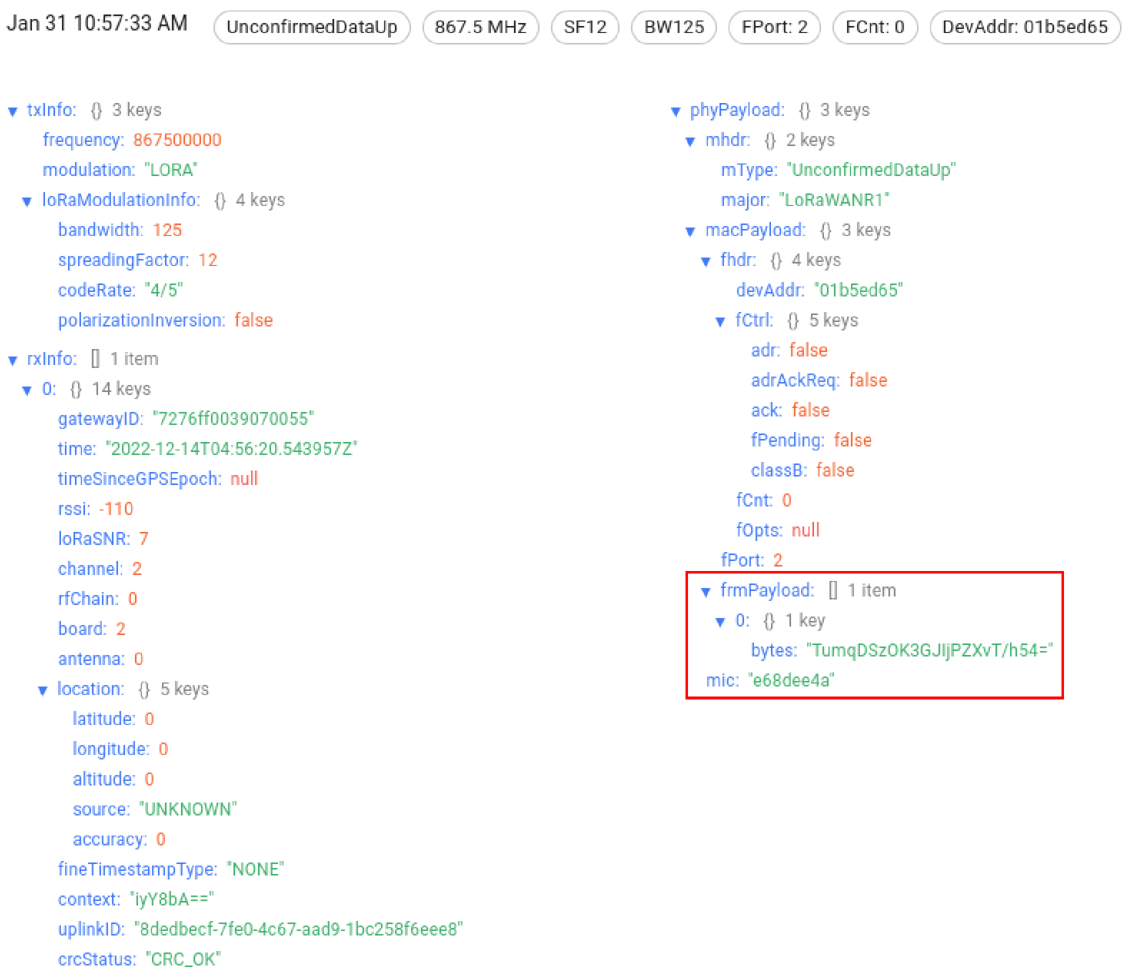
If we find "my message" on the server side, it means that we are able to transmit information from the endpoint to the server and get it back.

On the server, we have two tabs: device data and LoRaWAN frames. We can see these two tabs on figures 8 and 9.



```
▼ txInfo:  {}  3 keys
    frequency:  868100000
    modulation:  "LORA"
  ▼ loRaModulationInfo:  {}  4 keys
      bandwidth:  125
      spreadingFactor:  12
      codeRate:  "4/5"
      polarizationInversion:  false
  adr:  false
  dr:  0
  fCnt:  0
  fPort:  2
  data:  "bXkgbWVzc2FnZQ=="
  objectJSON:  ""
  tags:  {}  0 keys
  confirmedUplink:  true
  devAddr:  "005dee16"
```

Figure 8: Device data tab (interesting information emphasized with the red ellipse).



```
Jan 31 10:57:33 AM   UnconfirmedDataUp   867.5 MHz   SF12   BW125   FPort: 2   FCnt: 0   DevAddr: 01b5ed65

▼ txInfo:  {} 3 keys                                   ▼ phyPayload:  {} 3 keys
    frequency: 867500000                                 ▼ mhdr:  {} 2 keys
    modulation: "LORA"                                        mType: "UnconfirmedDataUp"
  ▼ loRaModulationInfo:  {} 4 keys                            major: "LoRaWANR1"
      bandwidth: 125                                     ▼ macPayload:  {} 3 keys
      spreadingFactor: 12                                  ▼ fhdr:  {} 4 keys
      codeRate: "4/5"                                          devAddr: "01b5ed65"
      polarizationInversion: false                           ▼ fCtrl:  {} 5 keys
                                                                 adr: false
▼ rxInfo:  [] 1 item                                             adrAckReq: false
  ▼ 0:  {} 14 keys                                               ack: false
      gatewayID: "7276ff0039070055"                              fPending: false
      time: "2022-12-14T04:56:20.543957Z"                        classB: false
      timeSinceGPSEpoch: null                                  fCnt: 0
      rssi: -110                                               fOpts: null
      loRaSNR: 7                                             fPort: 2
      channel: 2                                           ▼ frmPayload:  [] 1 item
      rfChain: 0                                             ▼ 0:  {} 1 key
      board: 2                                                   bytes: "TumqDSzOK3GJIjPZXvT/h54="
      antenna: 0                                           mic: "e68dee4a"
    ▼ location:  {} 5 keys
        latitude: 0
        longitude: 0
        altitude: 0
        source: "UNKNOWN"
        accuracy: 0
      fineTimestampType: "NONE"
      context: "iyY8bA=="
      uplinkID: "8dedbecf-7fe0-4c67-aad9-1bc258f6eee8"
      crcStatus: "CRC_OK"
```

Figure 9: LoRaWAN frame tab.

We can see on the figure 9 many information like the frequency, bandwidth, the frame counter, the message type...

Among these data, we find the frame paylaod *frmpayload* which is encrypted with the appKey.

We can decrypt it with the key but in the other tab on figure 8, we have like before the frequency, bandwidth, spreading factor... But also a data category which seems to be our decrypted message.

It took us some time to determine how to decode these data (how to read them) but finally we found that it was in base 64. Using a simple base 64 decoder, we found that "bXkgbWVzc2FnZQ==" becomes "my message" which is the message that we sent. So we are able to get back the information sent from the endpoint.

At this point, we conclude that we cannot implement the relay if the communication frequency between the endpoint and the gateway keeps changing each time a packet is emitted. Because the relay cannot listen to many frequencies, we need to have a good control over these communication frequencies. So we searched for a solution to reduce the communication between the endpoint and the gateway to only one frequency.

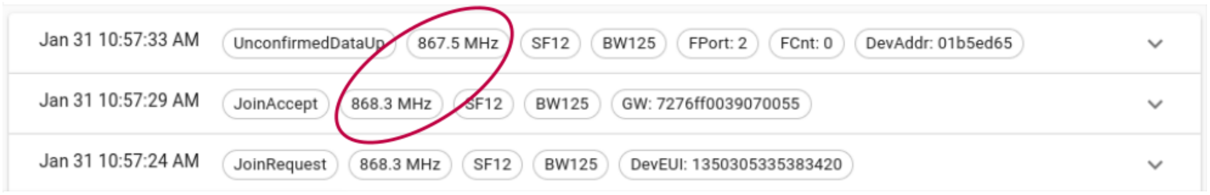
5 Reducing the endpoint communication to one frequency

5.1 Introduction

The objective is to reduce the communication of the endpoint and the relay to only one frequency.

LoRaWAN protocol uses many frequencies. In Europe for example, there are 3 default channels : 868.1MHz, 868.3MHz and 868.5 MHz. The others channels are 867.1MHz, 867.3MHz, 867.5MHz, 867.7MHz, 867.9MHz, 868.8MHz. They are divided in two sub-bands.

As we can see on the figure 10, most of these frequencies are used in the basic communication between endpoint and gateway.



```
Jan 31 10:57:33 AM   UnconfirmedDataUp   867.5 MHz   SF12   BW125   FPort: 2   FCnt: 0   DevAddr: 01b5ed65
Jan 31 10:57:29 AM   JoinAccept   868.3 MHz   SF12   BW125   GW: 7276ff0039070055
Jan 31 10:57:24 AM   JoinRequest   868.3 MHz   SF12   BW125   DevEUI: 1350305335383420
```

Figure 10: Basic communication between endpoint and gateway.

As explained before, we need to have the control of these frequencies in order to implement a relay.

In order to reduce the communication to one frequency, we considered three options that we will explain below.

5.2 Use a mask for the channels

The first solution considered is to put a mask on the frequency plan. We discovered that in RIOT library we have a function to apply a mask on frequency channels:

```
uint16_t mask = 0xF0;
semtech_loramac_set_channels_mask(&loramac,&mask);
```

With this function, we can apply a mask and use only a set of frequencies.

But the way it works is that we cannot choose exactly which frequency is used and which is unused. In the documentation we found that only two masks can normally be applied : 0x0f and 0xf0 (for the Europe frequency plan.)

These two masks are used to choose between the first or the second sub-band.

This is not a solution sufficient for our application, but it can be useful to at least limit to fewer frequencies. It can maybe be coupled with another solution to achieve our goal of one frequency.

5.3 Delete the channels definitions in header files

The second idea was to delete the channels in the header file.

We have a header file *region_EU868.h* where we find some channel definitions:

```
#define EU868_LC1 {868100000, 0, {((DR_5 << 4) | DR_0)}, 1}
#define EU868_LC2 {868300000, 0, {((DR_5 << 4) | DR_0)}, 1}
#define EU868_LC3 {868500000, 0, {((DR_5 << 4) | DR_0)}, 1}
```

We replace all frequencies by the same 868.1MHz (868100000 in the listing.)

This way, if the frequency does not exist, the communication cannot use these frequencies to send packets.

But this solution does not solve our problem because we continue to see several frequencies on the server.

The reason we found was that in the join accept message, the gateway provides a list of channel (CFList) to be used, and we did not succeed to change this CFList once received.

This lead us to the next idea: change randomness of channels.

5.4 Change the way to choose the next channel of communication

We have seen that channels are chosen randomly. If we change this randomness and replace it by something known, maybe we can solve our problem and have only one frequency.

In order to do that we first need to find where the next channel is chosen.

We found that the next channel is defined in *REGION_EU868.c* :

```
*channel = enabledChannels[randr( 0, nbEnabledChannels - 1 )];
```

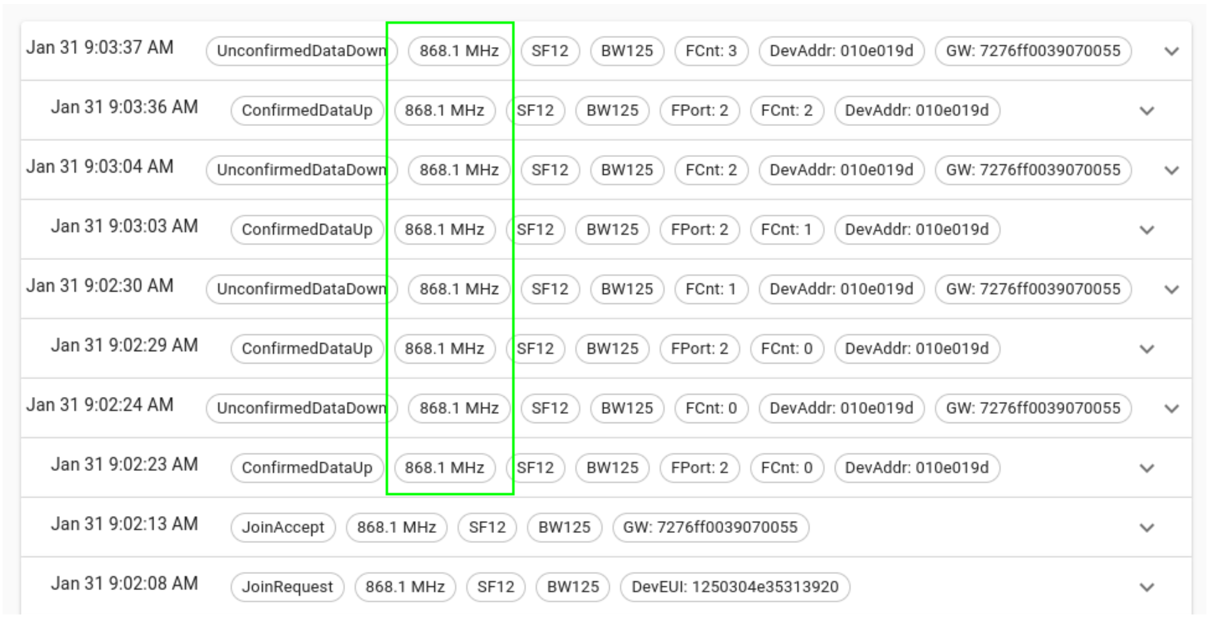
We replace this line by the following :

```
*channel = enabledChannels[1];
```

At first we took the first element of *enabledChannels* and we saw on the server that messages was sent alternatively between two frequencies 868.1MHz and 868.3MHz.

We concluded that we achieved to limit the frequencies to two channels using the previous tests. We also assumed that a channel used becomes available again and is placed at the end of *enabledChannels*. We change the code to take the second element of *enabledChannels* in order to have a LIFO stack method (Last In First Out).

This way we only have one frequency used to send the messages as we can see on figure 11.



```
Jan 31 9:03:37 AM  UnconfirmedDataDown  868.1 MHz  SF12  BW125  FCnt: 3  DevAddr: 010e019d  GW: 7276ff0039070055
Jan 31 9:03:36 AM  ConfirmedDataUp  868.1 MHz  SF12  BW125  FPort: 2  FCnt: 2  DevAddr: 010e019d
Jan 31 9:03:04 AM  UnconfirmedDataDown  868.1 MHz  SF12  BW125  FCnt: 2  DevAddr: 010e019d  GW: 7276ff0039070055
Jan 31 9:03:03 AM  ConfirmedDataUp  868.1 MHz  SF12  BW125  FPort: 2  FCnt: 1  DevAddr: 010e019d
Jan 31 9:02:30 AM  UnconfirmedDataDown  868.1 MHz  SF12  BW125  FCnt: 1  DevAddr: 010e019d  GW: 7276ff0039070055
Jan 31 9:02:29 AM  ConfirmedDataUp  868.1 MHz  SF12  BW125  FPort: 2  FCnt: 0  DevAddr: 010e019d
Jan 31 9:02:24 AM  UnconfirmedDataDown  868.1 MHz  SF12  BW125  FCnt: 0  DevAddr: 010e019d  GW: 7276ff0039070055
Jan 31 9:02:23 AM  ConfirmedDataUp  868.1 MHz  SF12  BW125  FPort: 2  FCnt: 0  DevAddr: 010e019d
Jan 31 9:02:13 AM  JoinAccept  868.1 MHz  SF12  BW125  GW: 7276ff0039070055
Jan 31 9:02:08 AM  JoinRequest  868.1 MHz  SF12  BW125  DevEUI: 1250304e35313920
```

Figure 11: Communication between endpoint and Gateway with only one frequency.

But this method is not satisfying.

First we found that if we activate the debug messages in *REGION_EU868.c*, the channel takes more time to become available again and the next channel is not the one that we expect.

In addition we did not achieve to choose exactly the frequency among the 8 of the Europe region. The only frequency that we succeed to "isolate" is 868.1MHz. Maybe if we change the channel definition to 868.3MHz instead of 868.1MHz it can be possible to switch to this other frequency.

However, it is enough to begin the implementation of a relay.

6 Implementing the relay

Now that we have all the tools in hands, we can implement our relay. To do so, we proceed step by step. The first step is to be able to send a message from the endpoint and receive it on the relay.

The principle of the relay is that it cannot decode messages passing through it. When a message is received from one side, it is just repeated on the other side (scheme on figure 12). So a join procedure is not needed by the relay and the endpoint or the gateway can ignore its existence.

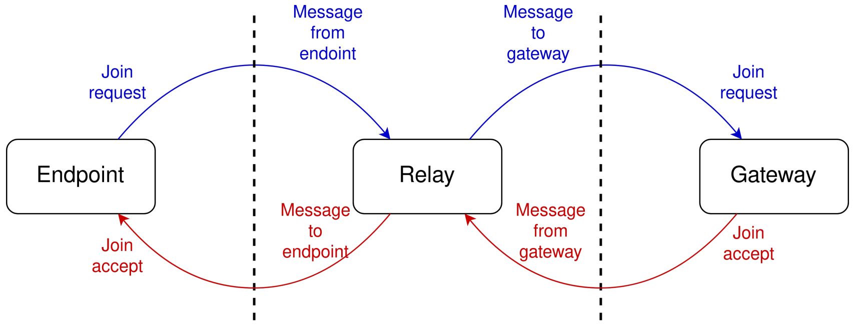


Figure 12: Scheme of the relay communications.

The essential need of the relay is that no "overlayer" must be added on the message. No key has to be generated by the relay. The message that enters the relay must be exactly the same that the one that go out of the relay.

6.1 Receive a message from endpoint on the relay

We already achieved to send a message from the endpoint to the gateway. Now we will try to send the message the same way but this time it is the relay that will listen to the message.

We will use the same configuration as before: the server is joined by the endpoint and messages are sent through the gateway. But we add the relay only receiving (not emitting) and we will try to receive the message. We can see the scheme of this configuration on the figure 13

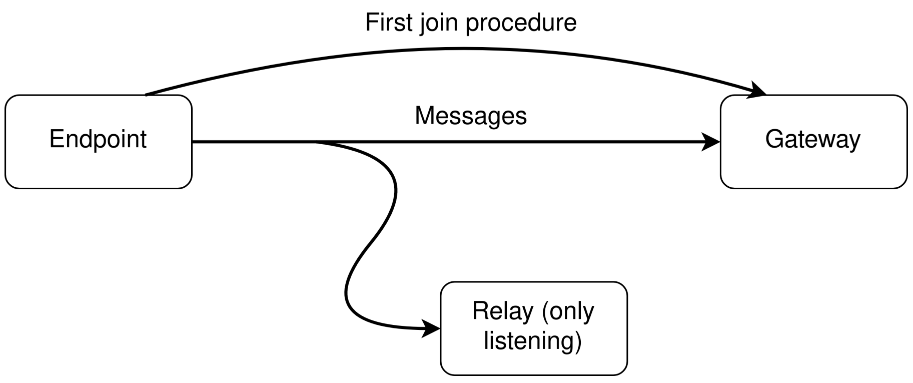


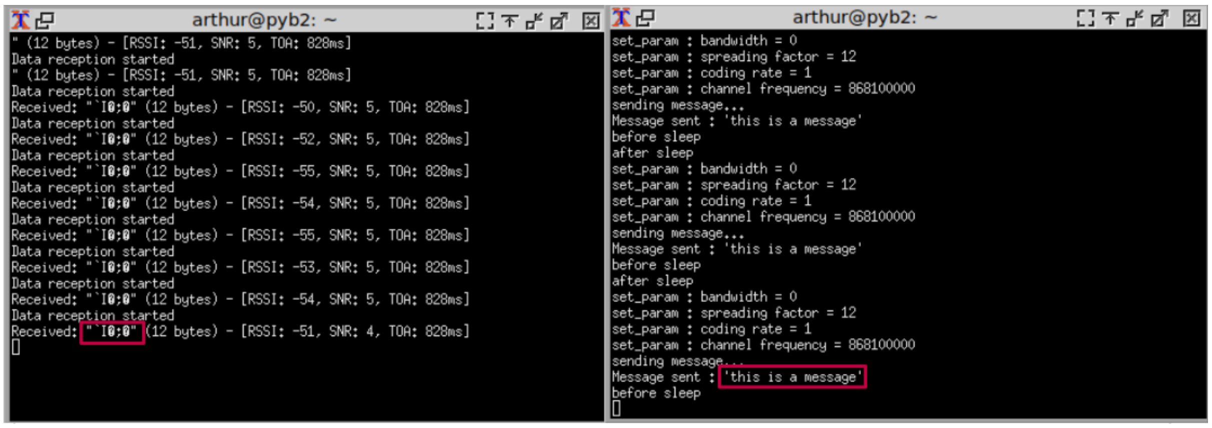
Figure 13: Scheme of the configuration where the messages of the endpoint addressed to the gateway are listened by the relay.

In order to do that, we must try to receive a message. When we communicated between the endpoint and the gateway, the endpoint reception was included in the RIOT function of join for the join procedure or the send function for the other messages.

To receive a message, we use the ISR (Interrupt Service Routine) to handle the reception with a function of event callback and a receive thread. We are inspired by the source code of the test of the driver sx126x in RIOT which also uses these function.[7]

When data are received, the function will fill a table. This table contains the message and can be displayed.

We can see an example of communications on the figure 14. We have at first some problems to correctly display the messages on the relay but this indicates at least that we receive something, always the same thing and each time we emit a message with the endpoint. This indicates that we get the right message but we must change the way we display it



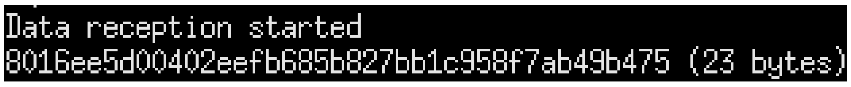
```
" (12 bytes) - [RSSI: -51, SNR: 5, TOA: 828ms]
Data reception started
" (12 bytes) - [RSSI: -51, SNR: 5, TOA: 828ms]
Data reception started
Received: "`I0;0" (12 bytes) - [RSSI: -50, SNR: 5, TOA: 828ms]
Data reception started
Received: "`I0;0" (12 bytes) - [RSSI: -52, SNR: 5, TOA: 828ms]
Data reception started
Received: "`I0;0" (12 bytes) - [RSSI: -55, SNR: 5, TOA: 828ms]
Data reception started
Received: "`I0;0" (12 bytes) - [RSSI: -54, SNR: 5, TOA: 828ms]
Data reception started
Received: "`I0;0" (12 bytes) - [RSSI: -55, SNR: 5, TOA: 828ms]
Data reception started
Received: "`I0;0" (12 bytes) - [RSSI: -53, SNR: 5, TOA: 828ms]
Data reception started
Received: "`I0;0" (12 bytes) - [RSSI: -54, SNR: 5, TOA: 828ms]
Data reception started
Received: "`I0;0" (12 bytes) - [RSSI: -51, SNR: 4, TOA: 828ms]
```

```
set_param : bandwidth = 0
set_param : spreading factor = 12
set_param : coding rate = 1
set_param : channel frequency = 868100000
sending message...
Message sent : 'this is a message'
before sleep
after sleep
set_param : bandwidth = 0
set_param : spreading factor = 12
set_param : coding rate = 1
set_param : channel frequency = 868100000
sending message...
Message sent : 'this is a message'
before sleep
after sleep
set_param : bandwidth = 0
set_param : spreading factor = 12
set_param : coding rate = 1
set_param : channel frequency = 868100000
sending message...
Message sent : 'this is a message'
before sleep
```

Figure 14: Communications between endpoint (at right) and gateway, relay is receiving the message (at left).

After some modifications, we are able to display correctly the packet and we can see the packet received on the figure 15. We notice that the message begins by *0x80* which means that the message is *confirmed data up*.

[7]functions at https://github.com/RIOT-OS/RIOT/blob/master/tests/driver_sx126x/main.c



```
Data reception started
8016ee5d00402eefb685b827bb1c958f7ab49b475 (23 bytes)
```

Figure 15: Packet received on the relay.

We are able to receive the message from the endpoint.

6.2 Sending a message from the relay and receive it on the endpoint

The next step is to be able to send a message with the relay. The message emitted must not be altered from the message received.

We will try to send a message and we program the endpoint to only receive (as the relay in the previous section). It is summarized by the scheme of the figure 16

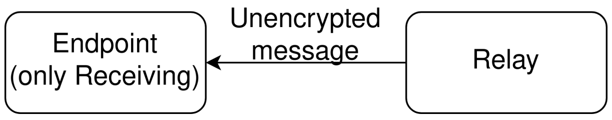


Figure 16: Scheme of the communication between the relay and the endpoint with the relay sending a message.

As we already said, the RIOT function of sending needs that a join procedure is done.

To send with the relay, we create our own function with netdev :

```
iolist_t iolist = {
        .iol_base = payload,
        .iol_len = len
    };
netdev->driver->send(netdev, &iolist)
```

This way we are not restricted by the join procedure.

Before sending, we must change the parameters to have the same between the endpoint and the gateway. We choose the channel frequency, the bandwidth, the spreading factor and the CRC (Cyclic Redundancy Check) with the RIOT functions to set parameters as we saw in the RIOT section 3.

Another parameter to change is the `NETOPT_IQ_INVERT`. It defines the direction of the chirp (up or down). This parameter is set to `NETOPT_ENABLE` or `NETOPT_DISABLE`, if it is *enable* the messages will only be seen by the gateway, if it is *disable* the messages will only be seen by the endpoint. We must for our test set it to disable if we want the messages to be received by the endpoint.

The messages are successfully received on the endpoint, but only the first time. Afterwards the next messages are not received by the endpoint.

After some investigations, we discovered that some parameters that we set are changed at another time than when we changed them. We did not find exactly when they are changed but to resolve this problem we always change the parameters before sending to be sure that they are at the correct values.

6.3 Communication from endpoint to gateway passing by the relay

Now that the relay can receive and send a message (from an endpoint and to an endpoint) we will try to do the complete communication with the gateway. The communication is represented by the scheme of the figure 17.

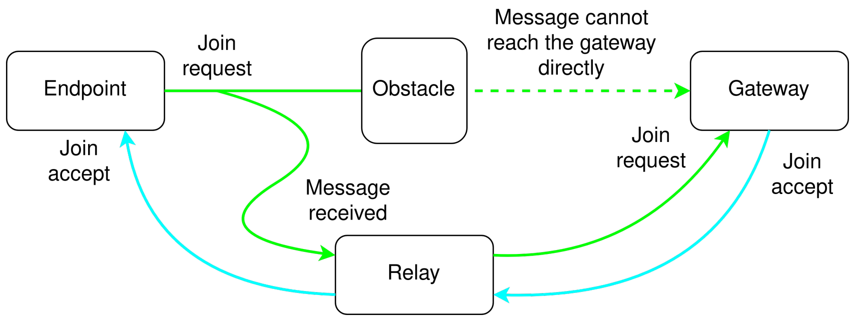
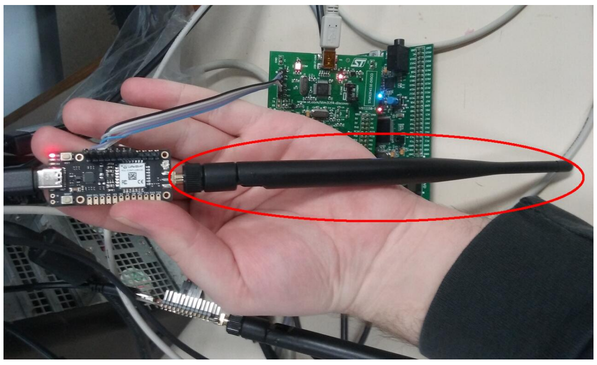


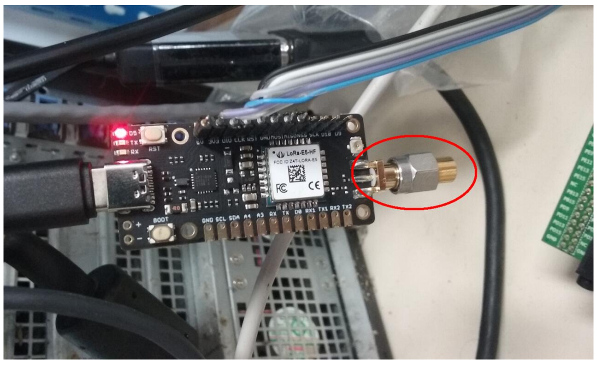
Figure 17: Scheme of the communications between endpoint, gateway and relay.

To simulate an obstacle, we remove the antenna of the LoRaE5 card and we connect a matched 50 ohms load termination. We can see the cards on the figure 18. Even after removing the antenna, the messages from the endpoint are still received by the gateway. We had to move the gateway (which was in another room) behind a desk.

In this configuration (figure 18b) the LoRaE5 card with an antenna (the relay) can communicate with the gateway and the other LoRaE5 card with a termination (the endpoint). The LoRaE5 card with a termination and the gateway can only communicate with the LoRaE5 card with an antenna.



(a) LoRaE5 card with an antenna (no need of relay).



(b) LoRaE5 card with a termination (a relay is needed).

Figure 18: Two configurations of the cards with an antenna (at left) and without (at right)

We can see the exchanges between the endpoint and the relay on the figure 19.

The "up" message from the endpoint to the gateway are indicated by the green rectangles. The "down" message from the gateway to the endpoint are indicated by the cyan rectangles. And the "direction" of the chirp are indicated by the red rectangles (0 is up and 1 is down).

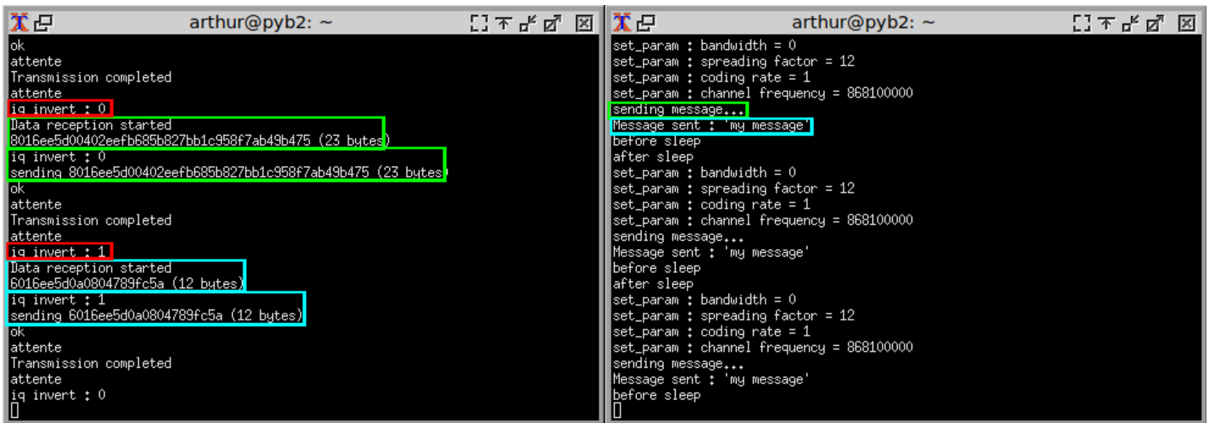
We can see on the right that a message is sent by the endpoint in green rectangle. The `iq_invert` parameter of the relay in red rectangle was set to 0 meaning that it is listening to incoming messages from the endpoint. The message is received by the relay on the left in the green rectangle, this packet begins by *0x80* it is a *confirmed data up* message.

This same message is then sent to the gateway. When the transmission is done, the `iq_invert` parameter of the relay is changed to 1 to listen messages from the gateway in the second red rectangle.

When the message from the gateway is received by the relay in the cyan rectangle, The same message is sent to the endpoint by the relay. This message begins by *0x60* that indicates that this is a *unconfirmed data down* message as we can see on the figure 6.

When the message is received by the endpoint at right in the cyan rectangle, the endpoint receive the confirmation that its message in the green rectangle has been correctly received. The endpoint indicates that the message has been sent.

The endpoint can now sleep, waiting to send the next message. The time between two messages is determined in the endpoint program.



```
arthur@pyb2: ~
ok
attente
Transmission completed
attente
iq invert : 0
Data reception started
8016ee5d00402eefb685b827bb1c958f7ab49b475 (23 bytes)
iq invert : 0
sending 8016ee5d00402eefb685b827bb1c958f7ab49b475 (23 bytes)
ok
attente
Transmission completed
attente
iq invert : 1
Data reception started
6016ee5d0a0804789fc5a (12 bytes)
iq invert : 1
sending 6016ee5d0a0804789fc5a (12 bytes)
ok
attente
Transmission completed
attente
iq invert : 0
```

```
arthur@pyb2: ~
set_param : bandwidth = 0
set_param : spreading factor = 12
set_param : coding rate = 1
set_param : channel frequency = 868100000
sending message...
Message sent : 'my message'
before sleep
after sleep
set_param : bandwidth = 0
set_param : spreading factor = 12
set_param : coding rate = 1
set_param : channel frequency = 868100000
sending message...
Message sent : 'my message'
before sleep
after sleep
set_param : bandwidth = 0
set_param : spreading factor = 12
set_param : coding rate = 1
set_param : channel frequency = 868100000
sending message...
Message sent : 'my message'
before sleep
```

Figure 19: Transmission between the relay (at left), the endpoint (at right).

This seems to work for a while, but at some point the message received by the relay begins by *0x40* which is a *unconfirmed data up* message according to the figure 6. This message is not sent by our endpoint because we do not see the endpoint trying to send a message and the message is a *unconfirmed data up* that is not the type of messages sent by our endpoint.

This is a problem because the gateway does not respond to this message, as no message was received by the relay, the relay is always listening the gateway for a response. The relay is blocked in that state and cannot listen the new messages of the endpoint.

To solve this problem we add a watchdog that will reset the card if it is blocked too long in a state. We use for this the functions seen in the section 3. It is not the most satisfying solution but at least the relay is never blocked in a state.

7 Conclusion

To conclude about this project, we managed to implement a LoRaWAN relay.

First of all, we achieved to understand the LoRaWAN communication.

We have differentiated LoRa and LoRaWAN as they are not on the same layer. LoRa defines the physical layer and LoRaWAN the upper layer of LoRa which is the MAC layer.

We managed to understand how the chirp spread spectrum modulation works. This modulation is used by LoRa to modulate its signals. In this modulation, the frequency of the signal is increased or decreased over the time

We then learnt how the architecture of a LoRaWAN network is done. A gateway is listening to many sensor nodes called endpoints. This gateway can listen to many frequencies at a time and when data from endpoints are received they are transmitted to the server through an internet connectivity.

After that we tried to understand the LoRaWAN packet format. We noticed some useful information such as the `MHDR` that indicates the message type. We also knew where the message was in the packet. With this packet, we also discovered that each message sent increased a counter, this counter must be synchronised between the transmitter and the receiver. We also noticed a `syncword` in the preamble of the packet, this `syncword` is used to differentiate many LoRaWAN networks.

Once the LoRaWAN communication was well understood, we tried to program a LoRaE5 card to do a simple communication with a gateway.

To do that we used RIOT OS which allows us to easily achieve a LoRaWAN communication in few lines. This environment provides us a lot of functions that allow us to use a high level of abstraction.

This RIOT OS was very helpful for our project but because of this high abstraction level, it is harder to use if the function of RIOT is not what we want.

We managed to do a simple communication between an endpoint and the gateway.

At this point we wanted to be sure that the message that we sent from the endpoint could be found back on the server. The message on the server is displayed in base 64 and after the conversion in ASCII, we managed to find the original message.

With this first communication, we observed that LoRa uses frequency hopping. We concluded that this was a problem for our relay because it can only listen to one frequency at a time. We then tried to reduce the communication to one frequency.

After many trials we achieved to reduce the communication to one frequency using multiple methods: a mask for the channels with the RIOT functions, delete the channels definitions in header files, and remove the randomness of the channel choice to control it.

Afterwards, we tried to implement the relay step by step. The first step was to be able to receive a message on the relay. Once we knew that we could receive a message, we tried to send a message to an endpoint that was listening the same way that the relay.

When we confirmed that we are able to receive and send from the relay, we tried to implement it in a communication between an endpoint and a gateway. In this configuration the gateway cannot be reached by the endpoint and vice versa.

Without the relay, no packet is received on the server, but with the relay we can see that all the packets that are emitted are received on the server.

In order to improve the robustness of our relay, we add a watchdog to be sure that the relay will not be blocked in any state.

We then noticed something, when the messages of the gateway are sent too fast after receiving the message from the relay, the `iq_invert` parameter of the relay cannot be changed fast enough to receive the messages from the gateway.

This message from the gateway never reaches the endpoint because it is not transmitted by the relay. The message sent initially is considered to have not reached its destination.

It is not really problematic because all the messages are received by the gateway but if we want a better robustness we must try to solve this problem. If the message from the gateway can be delayed for a few seconds, we thought that it can solve this problem.

Another problem that we thought but did not experiment is when there are multiple endpoints. If two endpoints communicate at the same time, the relay cannot support the two communications. Two solutions are considered, the first is to schedule the sending of the messages from the endpoints. The other solution is to add a *queue* system in the relay.

Finally, compliance with the duty cycle of the relay requires that it emits fewer messages than was initially received from multiple endpoints. Aggregating the messages from multiple endpoints might be an option but would require deciphering the messages and hence having access to the cryptographic keys, conflicting with the security principles of LoRaWAN.